

Let's write Snake game!

Tommaso Allevi

Who I am



Software Architect

surely not a UI designer

Tommaso Allevi



<https://www.linkedin.com/in/tommaso-allevi-a9979045/>



<https://twitter.com/AlleviTommaso>



<https://github.com/allevi> (allevi)

tomallevi@gmail.com

Node.js

React

SQL

Rust

Angular

Go

MongoDB

Python

PHP

Java

What is WASM



WebAssembly (abbreviated Wasm) is a binary instruction format for a **stack-based virtual machine**. Wasm is designed as a **portable compilation** target for programming languages, enabling deployment on the web for client and server applications.

Efficient and fast

Safe

Open and
debuggable

Part of the open
web platform

Core

Javascript API

Web API

WASI API

How to write WASM module?

WASM supports text and binary format, but probably you wouldn't write a module directly in WASM but use another language and compile it into WebAssembly.

Compile a WebAssembly module from:

- C/C++
- Rust
- AssemblyScript (a TypeScript-like syntax)
- C#
- F#
- Go
- ...

WebAssembly describes a memory-safe ...

Rust's rich type system and ownership model guarantee memory-safety ...



<https://www.rust-lang.org/it/what/wasm>

<https://www.rust-lang.org/>

<https://webassembly.org/getting-started/developers-guide/>

Which constraint we have on WASM in Rust?

- Only structs annotated with the attribute `# [wasm_bindgen]` are exported
- All exported method must be defined inside an `impl` block with `# [wasm_bindgen]`
- WASM knows only a few “base” types*: `bytes`, `intergers` (not all), `floating-points`, `vectors`)
- As consequence, enums need to be `# [repr (u8)]`
- As consequence, all returned types need to be converted

*<https://webassembly.github.io/spec/core/syntax/types.html>

The code



<https://github.com/allevo/snake-fosdem>

The code is a cargo workspace where you can find:

- `snake`: a Rust plain implementation of our game logic
- `handmade-snake`: Web Interface that wraps `snake` member
- `bevy-snake`: Bevy plugin for snake game built on top of `snake` member

The snake implementation

The crate exports

```
#[derive(Debug, Clone, Copy)]
4 implementations
pub enum Direction {
    Up,
    Down,
    Left,
    Right,
}
```

Direction is an enumeration for specify the direction

```
#[derive(PartialEq, Eq, Debug, Clone, Copy)]
5 implementations
pub struct Point {
    pub x: usize,
    pub y: usize,
}
```

A classic Point struct for tracking the position

```
pub struct Game { /* ... */ }

impl Game {
    pub fn tick(&mut self, mut direction: Direction) { /* ... */ }
    pub fn last_snapshot(&self) -> Snapshot { /* ... */ }

    pub fn dim(&self) -> (usize, usize) { /* ... */ }

    pub fn walls(&self) -> Vec<Point> { /* ... */ }
}

impl FromStr for Game {
    type Err = String;

    fn from_str(s: &str) -> Result<Self, Self::Err> { /* ... */ }
}
```

Game is the main struct that can be built from a string and has tick method to calculate the next state

```
#[derive(Debug, Clone)]
3 implementations
pub struct Snapshot {
    pub on_food: bool,
    pub on_wall: bool,
    pub eat_itself: bool,
    pub food_position: Point,
    pub snake: Vec<Point>,
    pub score: usize,
    pub period_duration: Duration,
}

impl Snapshot {
    pub fn get_dead_reason(&self) -> Option<&'static str> {
        if self.on_wall {
            return Some("On Wall");
        }
        if self.eat_itself {
            return Some("Eat itself");
        }
        None
    }
}
```

Snapshot is a struct that track a game state

How to use the snake implementation

```
static MY_LEVEL: &str = "\
#####
# f #
# h #
# b #
#   #
#####";

#[test]
fn test_ok() {
    let mut game: Game = MY_LEVEL.parse().unwrap();

    game.tick(Direction::Up);
    let snapshot: Snapshot = game.last_snapshot();
    assert_eq!(snapshot.on_food, true);
    assert_eq!(snapshot.on_wall, false);
    assert!(matches!(snapshot.get_game_over_reason(), None));

    game.tick(Direction::Up);
    let snapshot: Snapshot = game.last_snapshot();
    assert_eq!(snapshot.on_food, false);
    assert_eq!(snapshot.on_wall, true);
    assert!(matches!(snapshot.get_game_over_reason(), Some(_)));
}
```

a level definition
f = food
h = the snake head
b = the snake body

parse string and create game

play the game moving up the snake

get snapshot

check the exposed status

check if the game is over

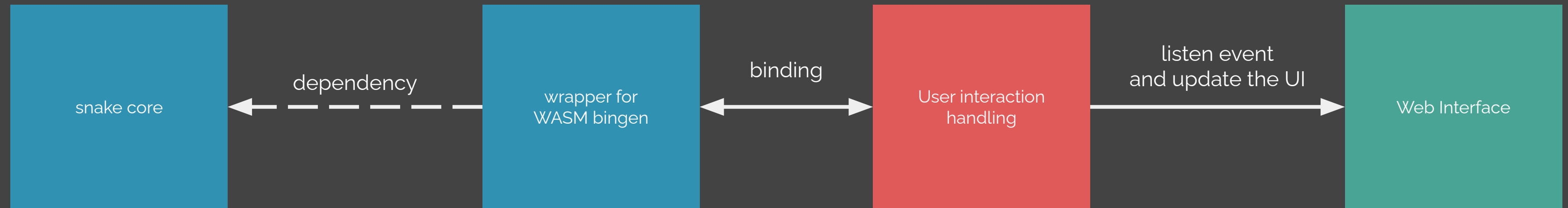
two public (non-test) snake level

```
pub static SNAKE_1: &str = "\
#####
#   #
#   #
# h  #
# b f #
#####";

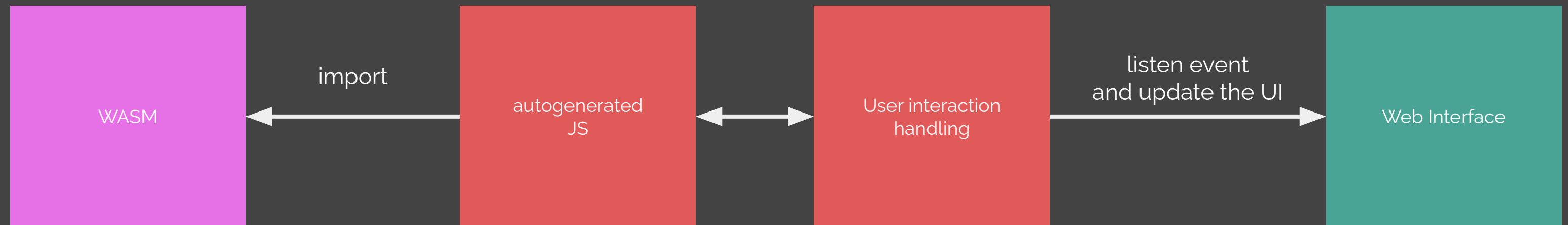
pub static SNAKE_2: &str = "\
h
b
| f |";
```


The conceptual approach

The code structure



The generated code after the compilation



rust code



js code



html code



wasm code

The handmade-snake code

```
#[wasm_bindgen]
#[repr(u8)]
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum DirectionWrapper {
    Up = 0,
    Down = 1,
    Left = 2,
    Right = 3,
}
```

DirectionWrapper duplicates the Direction enum implementation

```
#[wasm_bindgen]
pub struct PointWrapper(usize, usize);
```

The wrap of the Point struct

```
#[wasm_bindgen]
pub struct GameWrapper(Game);

#[wasm_bindgen]
impl GameWrapper {
    pub fn dim(&self) -> Int32Array { /* ... */ }

    pub fn walls(&self) -> Int32Array { /* ... */ }

    pub fn tick(&mut self, direction: DirectionWrapper) { /* ... */ }

    pub fn last_snapshot(&self) -> SnapshotWrapper { /* ... */ }
}
```

The wrap of the Game struct and the all wrapper methods that handle type conversion

```
#[wasm_bindgen]
pub struct SnapshotWrapper(Snapshot);

#[wasm_bindgen]
impl SnapshotWrapper {
    pub fn snake(&self) -> Int32Array { /* ... */ }

    pub fn food(&self) -> Int32Array { /* ... */ }

    pub fn get_game_over_reason(&self) -> JsValue { /* ... */ }

    pub fn score(&self) -> usize { /* ... */ }

    pub fn period_duration_ms(&self) -> usize { /* ... */ }
}
```

The SnapshotWrapper that wraps Snapshot struct and exposes all the member though a dedicated method

Unsustainable!
for this kind of project



DEMO

handmade-snake



A Bevy Engine introduction

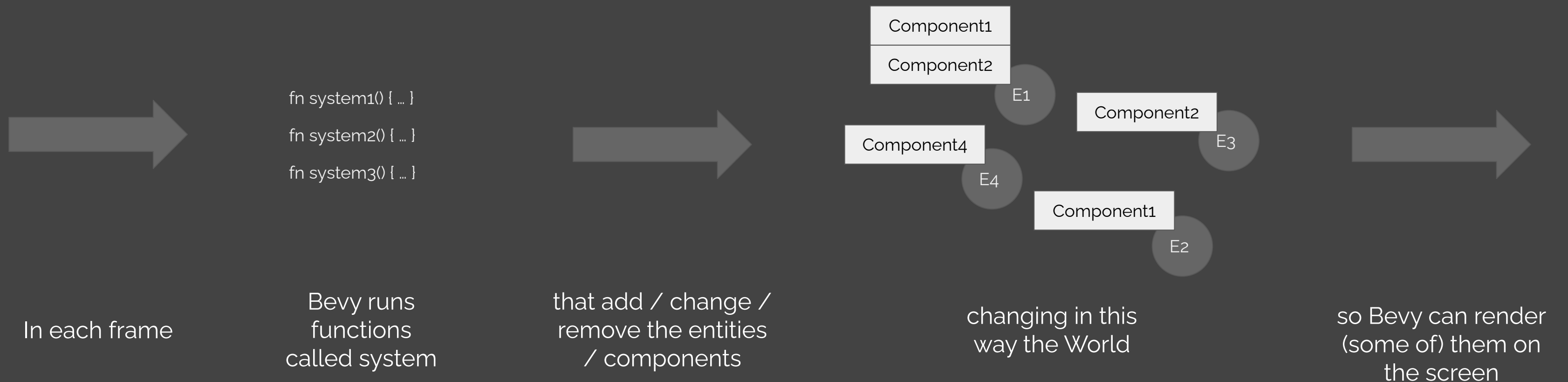
A refreshingly simple data-driven game engine built in Rust

Cross Platform: Windows, MacOS, Linux, Web, iOS and Android

ECS

Based on E(ntity)C(omponent)S(ystem) pattern:

- Entity: an object inside the system
- Component: a "tag" attached to an entity
- System: a function that works on the entities and components

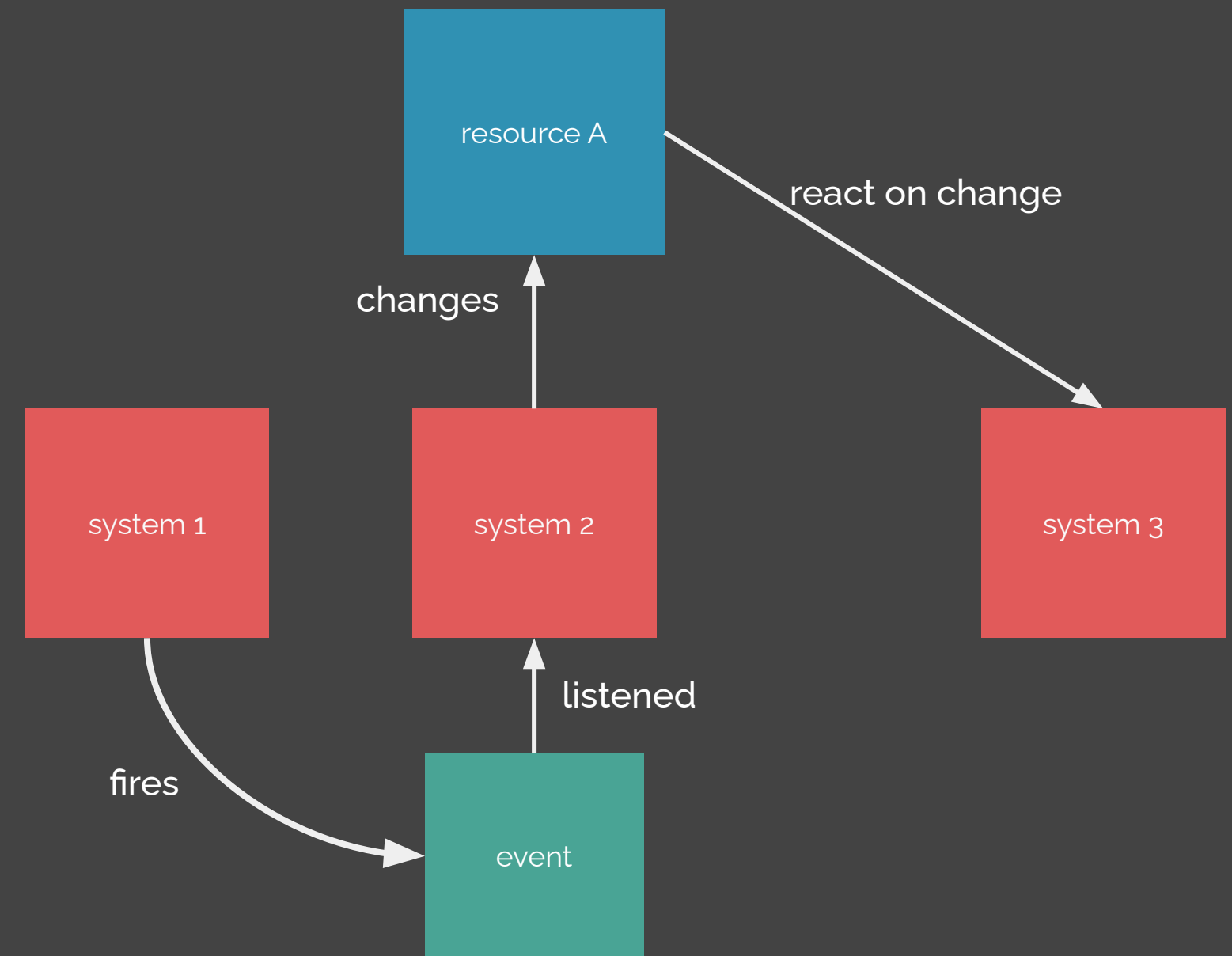


An introduction to the Bevy events and resources

An event* is a **Rust plain object** that can be fired / listened from systems that can react

A Resource** is a “**global instance**” stored inside Bevy engine and holds a Rust plain object. Bevy systems can listen a change on a resource

How do we use them?



*<https://bevy-cheatbook.github.io/programming/events.html>

**<https://bevy-cheatbook.github.io/programming/res.html>

An example of usage

The `tick` system example:

```
fn tick(  
    mut tick_event: EventReader<GameTick>,  
    mut game_over_writer: EventWriter<GameOver>,  
    mut game: ResMut<GameResource>,  
    mut score: ResMut<ScoreResource>,  
    mut snake: ResMut<SnakeResource>,  
    mut food_position: ResMut<FoodPositionResource>,  
    current_direction: Res<CurrentDirection>,  
    mut game_timers: ResMut<GameTimerResource>,  
) {  
    if tick_event.iter().count() == 0 {  
        return;  
    }  
  
    game.0.tick(direction: current_direction.0);  
  
    let snapshot: Snapshot = game.0.last_snapshot();  
  
    // game over  
    if let Some(reason: &str) = snapshot.get_game_over_reason() {  
        game_over_writer.send(event: GameOver(reason));  
        return;  
    }  
  
    // Update resources  
    snake.0 = snapshot.snake;  
    if score.0 != snapshot.score {  
        score.0 = snapshot.score;  
    }  
    if snapshot.food_position != food_position.0 {  
        food_position.0 = snapshot.food_position;  
    }  
    if game_timers.0.duration() != snapshot.period_duration {  
        game_timers.0.set_duration(snapshot.period_duration);  
    }  
}  
} fn tick
```

1. almost in all frame we did nothing: we are waiting for the "tick" event that is fired every X milliseconds

2. run tick

3. get game snapshot

4. check if the game is over

1. and fire the event

5. update the snake resource

6. update the score resource

7. update the food resource

8. update the duration of the timer

before going into the code

DEMO

bevy-snake

The structure of the application

We have 3 "state"s:

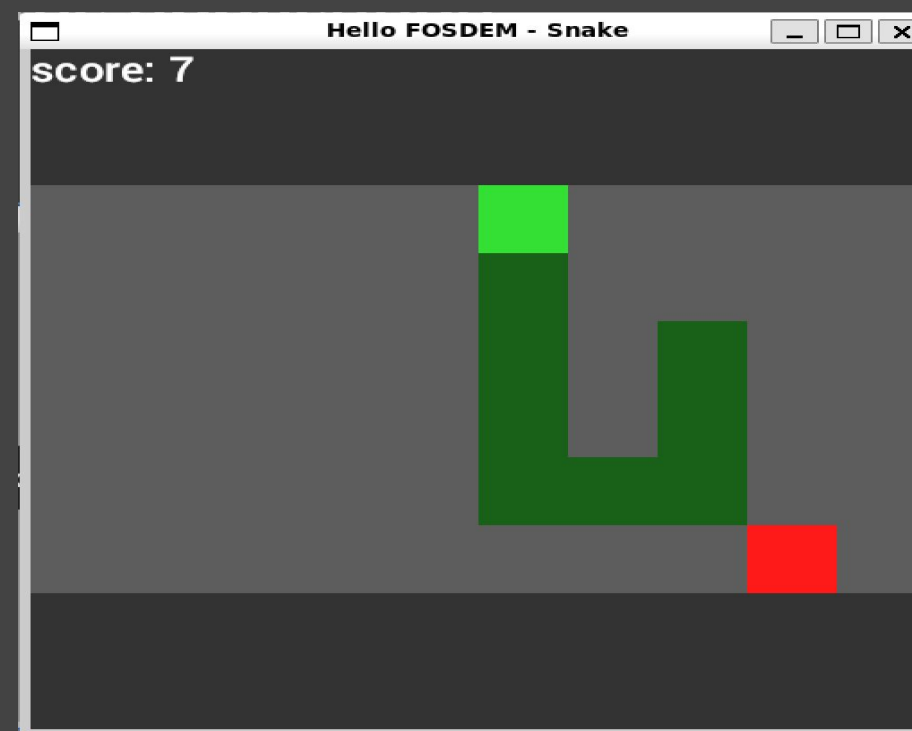
Choose game

`choose_game_plugin`



Play

`play_plugin`



Game Over

`game_over_plugin`



GameChosen
event

GameOver
event

Let's be focused on the `Play` state

When the user enters in `Play` state, we need to:

- create all the resources in bevy (`add_all_resources`)
- draw the initial status (`init_draw`)

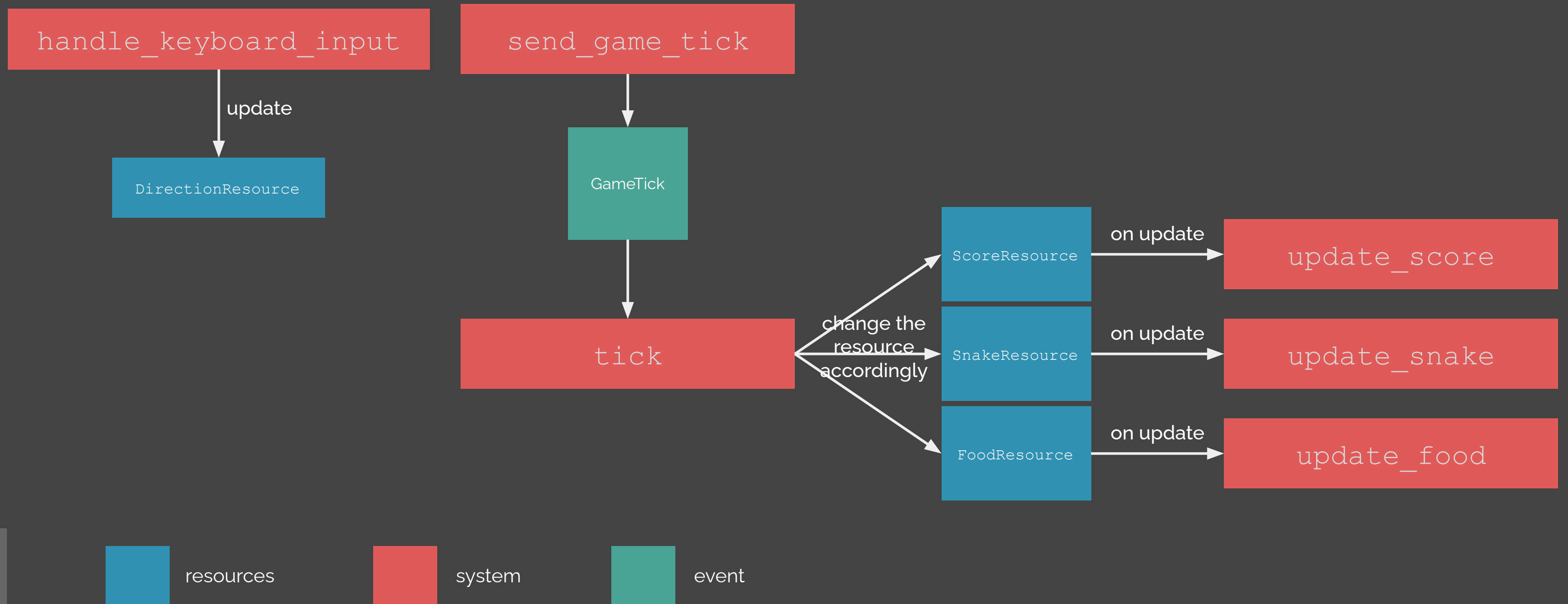
When the user stay in `Play` state, we need to:

- waiting the `GameTick` event, call `tick` game method and update the resources (`tick`)
- update the snake position (`update_snake`)
- update the food position (`update_food`)
- update the score number (`update_score`)

And:

- handling the pressed keys (`handle_keyboard_input`)
- fire the `GameTick` event (`send_game_tick`)

A graphical representation of the Play state systems



Show me the code!

Thank you



<https://github.com/allevo/snake-fosdem>